Fundamentos teóricos de los Paradigmas de Programación

Cátedra de Paradigmas de Programación
Facultad Regional Buenos Aires
Universidad Tecnológica Nacional

Coordinación:
Ing. Lucas Spigariol

Buenos Aires – Abril 2005

Presentación

¿Por qué paradigmas?

A lo largo de la historia de las ciencias de la computación han ido surgiendo herramientas, reglas, conceptos y otros elementos que permitieron la creación de los más variados lenguajes de programación. Algunos nacieron y al poco tiempo desaparecieron, mientras que otros hace años ya que se diseñaron y siguen vigentes. Muchos se fueron adaptando y renovándose para subsistir y se vuelven casi irreconocibles de sus orígenes y en cambio otros permanecen fieles a sus principios fundantes y casi inalterables, más allá de ciertos cambios cosméticos.

Los equipos, las metodologías y los campos de aplicación se transformaron radicalmente. Si Von Neumman se levantara de su tumba y se sentara frente a las computadoras de última generación se sentiría tan perdido como el inventor de la rueda al volante de un fórmula 1. Sin embargo, es probable que así como el conductor constataría que el vehículo sigue teniendo ruedas o preguntaría cómo hacen ahora para hacerla girar, los desarrolladores de software de otras épocas no dudarían de preguntar... ¿y cómo es la entrada de datos? ¿cómo los representan? ¿y el control de secuencia? ¿sigue existiendo el efecto de lado? Y la respuesta de un ingeniero en sistemas de esta época sería "Y... según el paradigma".

Si algo caracteriza al panorama actual del desarrollo de sistemas es su complejidad y heterogeneidad. No existe una sola forma de pensar y encarar las soluciones, no son uniformes los conceptos que fundamentan los lenguajes, no es única la manera de programar. Los modelos de datos, estructuras de control, mecanismos de evaluación, sentencias, enlaces, expresiones, declaraciones y tantos otros elementos que conforman los lenguajes de programación actuales son muy diferentes entre unos y otros, hasta opuestos, pero es posible detectar cuáles son los conceptos que marcan diferencias mayores o menores, muestran puntos de contacto o de inflexión, establecen criterios de clasificaciones, y así, aportan elementos teóricos para sistematizar el análisis de la programación.

De esta manera, tiene sentido hablar de la existencia de diferentes "paradigmas" de programación que aportan los fundamentos teóricos y conceptuales para desarrollar sistemas de una manera en particular, incluso podríamos decir con una "filosofía" especial, que los caracterizan, identifican y a la vez diferencian de los otros paradigmas.

En este trabajo presentamos un panorama general de los paradigmas de programación existentes en la actualidad y sus conceptos fundamentales.

En el primer capítulo tratamos sobre las características básicas de los lenguajes de programación y presentamos una clasificación global de los paradigmas.

Luego, en los dos capítulos siguientes, abordamos una serie de conceptos que son transversales a los diferentes paradigmas, y que hacen referencia a las formas de representar los datos y de organizar el control de la ejecución de los programas.

Tan importante como los conceptos individuales son las formas en que se relacionan para formar lenguajes de programación complejos. Por eso, presentamos, para terminar, un capítulo para cada uno de los paradigmas más importantes de la actualidad, con sus principales características.

El presente material fue elaborado con el aporte de un conjunto de docentes de la Facultad Regional Buenos Aires de la Universidad Tecnológica Nacional, basándonos en bibliografía especializada y en la experiencia de años de llevar adelante materias troncales de programación en la carrera de Ingeniería en Sistemas de Información.

Le deseamos que sea una herramienta de utilidad para su formación permanente y su ejercicio profesional.

Ing. Lucas Spigariol

Capítulo 1

Paradigmas de programación

- **Lenguajes**
- **Traductores e intérpretes**
- **Métodos de diseño de programas**
- **Surgimiento de los paradigmas**
- **Clasificación de los paradigmas**
  - Procedimentales
  - Declarativos
- **Principales paradigmas**

Lenguajes

Un lenguaje de programación es un lenguaje que permite establecer una comunicación entre el hombre y la máquina. Los lenguajes contienen un conjunto variado de comandos, instrucciones, reglas, formas de organizar los datos y otros elementos, que permiten codificar programas con los que se indican las tareas que la máquina debe realizar para procesar un conjunto de información determinado y resolver un problema.

Para ello, en los lenguajes existe una sintaxis (las reglas que hacen a las formas) y una semántica (los significados).

- La **sintaxis** de un lenguaje tiene que ver con la forma de los programas, es decir, con cómo deben escribirse las expresiones, los comandos, las declaraciones, y todo el código del programa para que sea entendible sin ambigüedades.
- La **semántica** de un lenguaje se relaciona con el significado de los programas, con cómo deben comportarse cuando son ejecutados en una computadora.

Traductores e intérpretes

Un programa escrito mediante un lenguaje de programación, se llama fuente o simbólico. Pero éste programa fuente no es ejecutable, pues a la máquina le es ininteligible, requieren de un paso previo que implica la conversión del lenguaje artificial al lenguaje absoluto (o de máquina), que sí sea ejecutable.

Los conversores de lenguaje fuente a ejecutable se pueden clasificar en traductores e intérpretes.

Traductores

Los **traductores** convierten el programa fuente a una versión equivalente en lenguaje de máquina y dejan "armada" esta versión (programa objeto) para que el computador la ejecute cuantas veces sea necesario sin que el traductor vuelva a intervenir.

Los traductores que producen un código ejecutable (un programa objeto) que luego es ejecutado por el computador en forma directa cuantas veces se quiera. Son de dos tipos: ensambladores, también llamados compaginadores, y compiladores.

Un **ensamblador** es un programa que traduce un programa fuente escrito en un tipo especial de lenguaje, "assembler", al lenguaje de máquina. La estructura del "assembler" es similar al lenguaje de máquina, código de operación y operandos, pero en lugar de la notación hexadecimal se utilizan códigos mnemotécnicos. El programa ensamblador lee el programa fuente escrito con simbólicos, traduce los códigos de operación a códigos de operación en lenguaje de máquina y asigna direcciones de memoria para almacenar las instrucciones y los datos. Cada instrucción simbólica se traduce en una instrucción en lenguaje de máquina y por esta característica se lo llama lenguaje de **bajo nivel**.

Los otros tipos de traductores son los **compiladores**. Estos traductores generan por cada sentencia del lenguaje fuente muchas instrucciones de lenguaje de máquina. Se los llama lenguajes de **alto nivel.**

Un programa escrito en un lenguaje de bajo nivel es mucho más rápido que otro semejante, pero escrito en un lenguaje de alto nivel y luego traducido con un compilador, pues utiliza la menor cantidad de instrucciones de máquina necesarias para lograr el objetivo. El absoluto producido por un compilador tiene más instrucciones pues cada sentencia se expande en una cantidad de instrucciones de máquina. Esto se produce en tiempo de compilación y el programador no tiene control sobre la traducción, a lo sumo puede llegar a conocer que código es más eficiente. Por otra parte, los lenguajes de bajo nivel presentan mayor dificultad para depurar un programa, modificarlo y mantenerlo.

Intérpretes

Los **intérpretes**, a diferencia de los traductores, ejecutan las instrucciones del programa fuente, interpretando cada vez las sentencias del programa. Va traduciendo y ejecutando cada vez que el programa lo requiere, y no almacena programa objeto alguno. Lo que se almacena es el interprete que reside en memoria, el programa fuente, tablas de símbolos y datos necesarios para la ejecución del programa.

La modalidad interpretativa tiene como ventaja sobre la traducción la posibilidad de poner a punto los programas con más rapidez, ya que entre prueba y prueba no hay que pasar por la etapa de una nueva compilación o traducción. Como desventaja en cambio, la interpretación es más lenta ya que cada vez que se debe ejecutar el programa debe traducir cada vez, cada una de las instrucciones que se ejecutan.

En realidad los casos de interpretación pura o traducción pura son extremos. En la práctica y con el devenir del tiempo y el incremento de la velocidad de los procesadores, muchos lenguajes se implementan mediante una combinación de ambas técnicas o una traducción a un código intermedio que luego es interpretado.

Métodos de diseño de programas

Los métodos de diseño influyen en los lenguajes en el sentido de establecer condiciones que el lenguaje debe cumplir. La arquitectura ejerce una influencia en el sentido contrario, es decir, tiende a limitar el lenguaje para que éste se pueda implementar eficientemente en las máquinas actuales.

Una técnica de diseño y desarrollo de sistemas común es la modularización, que consiste en descomponer el sistema en módulos, que provocan un ocultamiento de la información. Cada módulo esconde información, tiene su funcionalidad y sus correspondientes estructuras de datos. Los módulos proporcionan formas de interactuar e intercambiar información según los lenguajes.

Hay dos grandes tendencias para orientar los diseños que son BOTTON – UP (ir de lo particular a lo general) o TOP – DOWN (de lo más general, ir descomponiendo hacia abajo, a lo más particular).

Surgimiento de los paradigmas

La historia del desarrollo de los lenguajes de programación muestra una creciente evolución en la que se van incorporando elementos que permiten ir creando programas cada vez más sólidos y eficientes y a la vez facilitar la tarea del programador para su desarrollo, mantenimiento y adaptación.

Sin embargo este proceso está lejos de ser lineal. Ciertas características básicas incorporadas por algunos lenguajes fueron puestas en duda y fuertemente criticadas por otros[1], de manera que se fueron perfilando diferentes teorías y grupos de lenguajes que postulaban formas disímiles de construir las soluciones.

Así, a medida que fueron avanzando las ciencias de la computación surgen lo que actualmente definimos como "paradigmas"

Un **paradigma de programación** es un **modelo** básico de diseño e implementación de programas. Un modelo que permite desarrollar programas conforme a ciertos principios o fundamentos específicos que se aceptan como válidos. En otras palabras, es una colección de modelos conceptuales que juntos modelan el proceso de diseño, orientan la forma de pensar y solucionar los problemas y, por lo tanto, determinan la estructura final de un programa.

Clasificación de los paradigmas

A los paradigmas se los podría clasificar de diversas maneras según los criterios que se prioricen. Pero partiendo de los principios fundamentales de cada paradigma en cuanto a las orientaciones sobre la forma para construir las soluciones, podemos distinguir entre los procedimentales y los declarativos.

- Paradigmas **procedimentales** u operacionales. Indican el modo de construir la solución, es decir detallan paso a paso el mecanismo para obtenerla.[2]
- Paradigmas **declarativos**. Describen las características que debe tener la solución. Es decir especifican "qué" se desea obtener pero no requieren indicar "cómo" obtenerla.

Existen también otros paradigmas, que no se pueden encuadrar en esta clasificación como el heurístico y el concurrente.

Por otra parte, hay autores que hacen otro tipo de enumeraciones o clasificaciones de paradigmas en base a criterios diferentes, por ejemplo como de "alto nivel" o "bajo nivel", o que subclasifican estas categorías en otras más específicas o acotadas, como por ejemplo "Paradigma orientado a eventos".

[1] Por ejemplo, durante años, los lenguajes han estado restringidos por las ideas de Von Neumann ya que la mayoría de las computadoras tienen una estructura muy similar a la original de éste. Se ha combatido este punto de vista critican el concepto de variables y de sentencias de asignación, diciendo que son la raíz de muchos males y una maldición transmitida por la arquitectura de Von Neumann.

[2] Una variante de los paradigmas procedimentales son los paradigmas demostrativos, que especifica la solución describiendo ejemplos y permitiendo que el sistema generalice la solución de estos ejemplos para otros casos. Aunque es fundamentalmente procedimental, existen autores que lo clasifican como una categoría separada: Ver ALONSO AMO, F y SEGOVIA PEREZ, F. *Entornos y Metodologías de Programación*. Capítulo 1.

Paradigmas procedimentales

También llamados operacionales, la característica fundamental de estos paradigmas es la **secuencia computacional** realizada etapa a etapa para resolver el problema.

Los programas realizados con lenguajes procedimentales deben incluir en su codificación las **instrucciones de control** para determinar el flujo de la ejecución, como decisiones, iteraciones y otras, conformando, de esta manera, diferentes "algoritmos".

Actúan modificando repetidamente la representación de sus datos, basándose en **asignaciones destructivas** con **efecto de lado**. Utilizan un modelo en el que las variables están estrechamente relacionadas con direcciones de la memoria del ordenador. Cuando se ejecuta el programa, el contenido de estas direcciones se actualiza repetidamente, pues las variables reciben múltiples asignaciones, y al finalizar el trabajo, los valores finales de las variables representan el resultado.

Su mayor dificultad reside en determinar si el valor computado es una solución correcta del problema, por lo que se han desarrollado multitud de técnicas de depuración y verificación para probar la corrección de los problemas desarrollados basándose en este tipo de paradigmas.[3]

En otras palabras, se basan en **"cómo"** lograr la solución.

Paradigmas declarativos

Los paradigmas declarativos se basan en desarrollar programas especificando o "declarando" un conjunto de proposiciones, condiciones, restricciones, afirmaciones, ecuaciones o transformaciones que caracterizan al problema y **describen su solución**.

A partir de esta información el sistema utiliza **mecanismos internos de control** que evalúan y relacionan adecuadamente dichas especificaciones, de manera de obtener la solución. No se necesita de la puntualización de los pasos a seguir para alcanzar una solución, ni instrucciones de control que conformen algoritmos. [4]

Estos paradigmas permiten utilizar variables para almacenar valores intermedios, pero no para actualizar estados de información. Si bien sus variables se relacionan con posiciones de memoria, no existe el concepto asignaciones destructivas. Las variables son usadas en expresiones, funciones o

[3] ALONSO AMO, F y SEGOVIA PEREZ, F. *Entornos y Metodologías de Programación*. Capítulo 1.

[4] En la práctica, las implementaciones lenguajes de los paradigmas declarativos, por motivos de eficiencia u otros, incorporan mínimamente algún tipo de secuencia de control por lo que, siendo estricto en los términos, se denominan lenguajes "pseudodeclarativos", aunque por simplicidad se los llama en general declarativos.

procedimientos, se unifican con diferentes valores, pero siempre con **transparencia referencial**, es decir, sin efecto de lado.

Como estos paradigmas especifican la solución sin indicar cómo construirla, en principio, eliminan la necesidad de probar que el valor calculado es el valor solución.

En otras palabras, se basan en **"qué"** es necesario especificar.

Principales paradigmas

Paradigmas procedimentales:

- Imperativo
- Objetos

Paradigmas declarativos:

- Funcional
- Lógico

Otros paradigmas:

- Heurístico
- Concurrente

Capítulo 2

Abstracción de Datos

- **Datos y valores**
- **Tipos de datos**
- **Clasificación de los tipos de datos**
  - Tipos de datos simples
  - Tipos de datos compuestos
  - Tipos de datos recursivos
- **Sistemas de tipos**
  - Fuertemente tipados
  - Débilmente tipados
- **Transparencia referencial y efecto de lado**

Datos y valores

Los *datos* son la materia prima de la computación y son los que dan sentido a la existencia de programas que los procesen. Entendiendo un sistema como una abstracción de la realidad, los datos son los elementos que representan cada una de los aspectos de la realidad que son significativos para el funcionamiento del sistema.

Cada dato implica un determinado valor para que tenga sentido como abstracción de la realidad. Para ser utilizado en un sistema, requiere de una convención que permita representar sin ambigüedad su valor. Así, puede ser operado, evaluado, almacenado, incorporados en una estructura de datos, pasadas como argumento a una función o procedimiento, ser devueltas como resultado de una función, etc.

Tipos de datos

El gran volumen y variedad de valores, hace necesario una agrupación o clasificación de los valores según sus semejanzas y diferencias. Para hacerlo se establecen lo que se denominan **"tipos de datos"**.

Si se toma como criterio las similitudes en cuanto al contenido de lo que representan, se puede aproximar una primera definición de tipo: *un tipo es una agrupación o conjunto de valores*. Por ejemplo: *{true, false}* es el conjunto que representa valores de verdad y se lo puede denominar como un tipo de dato "booleano". Sin embargo, no puede afirmarse que cualquier conjunto arbitrario de valores sea un tipo. Una condición que se debe tener en cuenta es que los valores dentro del tipo se comporten en forma uniforme respecto a cierto conjunto de operaciones. Por ejemplo, *{true, false}* puede ser considerado un tipo de dato ya que los valores true y false se comportan en forma uniforme respecto de las operaciones lógicas Y, O y NOT; mientras que *{1, hola, true}* no es un tipo ya que no se puede operar en forma uniforme con estos valores.

Se puede concluir, entonces, que un tipo de dato es un **conjunto de valores** más las **operaciones asociadas** a ellos.

Clasificación de los tipos de datos [5]

Al clasificar los tipos de datos tomando como criterio la estructura de representación de sus datos, se puede distinguir tipos de datos **simples** y **compuestos**, destacando entre los últimos, los tipos de dato **recursivos**.

La clasificación de un tipo de dato en particular, depende del lenguaje en que se analice, por lo que un tipo de dato puede, por ejemplo, ser implementado como tipo simple en un lenguaje y como tipo compuesto en otro, o ser un tipo recursivo en uno y en otro no.

Por ejemplo, un tipo de dato "cadena", comúnmente denominado "string", representa una secuencia de caracteres. Muchos lenguajes lo soportan, pero no existe un consenso en su estructura. En algunos de ellos puede ser entendido como tipos de datos simples, sin posibilidad de subdivisión y en otros como tipo compuesto, conformado por una estructura donde se puede identificar a cada uno de los caracteres que lo componen en forma independiente. Entre estos últimos, pueden definirse utilizando una estructura de tipo "arreglo" o "array", o con una estructura recursiva de "lista".

Tipos de datos simples

Un tipo de dato simple es aquél cuyos valores son **atómicos**, es decir, que no pueden ser descompuestos en otros valores, por lo que también es denominado como tipo de dato "atómico".

Los tipos simples más comunes son los siguientes:

- Booleanos = {true, false}

[5] Puede encontrarse una descripción más detallada en WATT, David. *Programming Languajes Concepts and Paradigms*.. Capítulo 2.2 a 2.4.

- Enteros = {…, -2, -1, 0, +1, +2, …}
- Reales = {…, -1.5 , …, 0.0, …, +1.2, …}
- Caracteres = {…, 'a', 'b', …, 'z', …}

Tipos de datos compuestos

Un tipo compuesto es un tipo de dato cuyos valores están **compuestos por otros valores**, de manera que conforma una estructura de datos. Cada uno de los valores que forman el tipo de dato compuesto corresponde a algún tipo de dato que puede ser tanto simple como compuesto. Su utilidad consiste en que se pueden procesar en su conjunto como una unidad o se pueden descomponer en sus partes y tratarlas en forma independiente.

Los lenguajes de programación en general tienen una amplia variedad de tipos de datos compuestos y sus múltiples combinaciones:

- Tuplas
- Registros
- Uniones
- Arreglos
- Árboles
- Listas

Tipos de datos recursivos

Entre los tipos de datos compuestos merecen una especial distinción los denominados tipos recursivos.

Un tipo de dato recursivo es un tipo compuesto en el que alguno o algunos de los valores que lo compone son, a su vez, del mismo tipo. Es decir, un tipo recursivo **se define en función de sí mismo**.

Algunos tipos recursivos son:

- Árboles
- Listas

Por ejemplo, una lista es una secuencia de valores que puede tener cualquier cantidad de elementos, incluyendo la posibilidad de no tener ningún elemento, caso en que es llamada "lista vacía". Una definición del tipo de dato "lista" puede ser una estructura de dos componentes, donde la primera es un dato de algún tipo en particular, que corresponde al contenido de la lista, y la segunda es del mismo tipo "lista" que se está definiendo. Además, el tipo de dato lista debe definir un valor que represente la lista vacía, para permitir ponerle fin a la recursividad y obtener listas de alguna longitud determinada.

Sistemas de tipos

Según la necesidad o predominancia que tenga un lenguaje sobre la definición de los tipos de datos que utiliza, se los clasifica en "fuertemente" o "débilmente" tipados.

Fuertemente tipados

En un lenguaje fuertemente tipado, toda variable y parámetro deben ser definidos de un tipo de dato en particular que se mantiene sin cambios durante la ejecución del programa.

Débilmente tipados

En un lenguaje débilmente tipado, las variables y parámetros no requieren ser definidas de algún tipo de dato en particular, sino que pueden asumir valores y tipos de datos diferentes durante la ejecución del programa.

Transparencia referencial y efecto de lado

El concepto de **transparencia referencial,** sintéticamente, consiste en que el valor de una expresión depende únicamente del valor de sus subexpresiones. El **efecto de lado**, también conocido como efecto colateral, hace referencia a lo contrario.

Los lenguajes con efecto de lado utilizan un modelo en el que las variables están estrechamente relacionadas con direcciones de la memoria del computador, y actúan modificando repetidamente la representación de sus datos, mediante **asignaciones destructivas**. De esta manera, el valor de una expresión puede depender de otros factores, como expresiones evaluadas anteriormente que modifican las posiciones de memoria, las evaluaciones previas de la misma expresión, los valores de otras variables no explicitadas en la invocación, etc.

El efecto de lado es propio de los lenguajes de los paradigmas procedimentales y es una de las características que los diferencian de los lenguajes de paradigmas declarativos.

La transparencia referencial, hace que las expresiones mantengan sus propiedades matemáticas, que se perderían en presencia del efecto de lado, por ejemplo, al asignar a una variable el resultado de otra e incrementar la segunda a la vez.

Capítulo 3

Abstracción de Control

- **Enlace**
  - Enlace estático
  - Enlace dinámico
- **Ámbito y entorno**
- **Pasaje de Parámetros**
  - Por copia
  - Por referencia
- **Mecanismos de evaluación**
  - Evaluación ansiosa
  - Evaluación diferida

Enlace

Un concepto común a todos los lenguajes de programación es la posibilidad de enlazar **identificadores a entidades**, que implica poder relacionar los valores con los procedimientos u operaciones que los utilizan.

Por ejemplo, las expresiones 'n + 1', 'func (x, 1)' o 'x = n' no pueden ser interpretadas aisladamente sino que se debe tener en cuenta qué representan los identificadores que en ellas aparecen. De alguna manera, el sistema debe poder reconocer cuál es la entidad identificada con las variables 'n' o 'x', en general su valor y tipo de dato, y determinar las entidades identificadas por '+', 'func' o '=', correspondientes en general a procedimientos u operaciones, para chequear que sea posible su ejecución y en caso afirmativo poder llevarla a cabo correctamente.

Las diferencias entre lenguajes están en función del momento, y en consecuencia la forma, en que se produce este enlace.

Enlace estático

En el **enlace estático**, el enlace entre identificadores y entidades se produce en **tiempo de compilación**. Cada variable, parámetro, procedimiento y en

general cualquier expresión evaluable, predefinida de un tipo de dato al programarse se chequea y, si resulta satisfactorio, se enlaza, al momento de compilarse un programa.

Sus principales ventajas son:

- **Depuración de errores***:* El chequeo en tiempo de compilación permite detectar fácilmente cierta cantidad de errores de codificación provocados por inconsistencias de tipos de datos en las expresiones, en vez de provocarse las fallas recién al momento de ejecutarse.
- **Eficiencia***:* En general, si bien depende de las implementaciones de cada lenguaje, su ejecución es más rápida al no requerir que se realicen cada vez los controles y enlace ya resueltos previamente, y por única vez, durante la compilación.

Enlace dinámico

En el **enlace dinámico**, el enlace entre identificadores y entidades (por ejemplo, entre el nombre de un procedimiento y el código que debe ejecutarse al ser invocado) tiene lugar en **tiempo de ejecución**. Al desarrollar un programa, las expresiones no son predefinidas de un tipo de dato en particular y, por lo tanto, no se chequean en tiempo de compilación, sino que el enlace se produce durante la ejecución del programa, de acuerdo a los valores que vayan asumiendo los elementos de cada expresión.

Sus principales ventajas son:

- **Extensibilidad**: El mismo identificador se aplica a distintas entidades en distintos lugares del código, y en distintos momentos de la ejecución de un programa.
- **Desarrollo de un código más claro y compacto***:* Se puede generalizar abstrayéndose de las particularidades propias de cada tipo de dato, puntualizando sólo en lo específico de cada caso y evitando la ramificación de construcciones condicionales.
- **Reusabilidad***:* Frente a cambios en la realidad que el sistema pretende modelar, impensables incluso al momento de su desarrollo, una solución pensada para cierto tipo de datos puede ser reutilizada para otro conjunto de datos, sin necesidad de recodificar y volver a compilar.

Ámbito y entorno

Muchos lenguajes permiten que un identificador dado se defina en varias partes de un programa, posiblemente denotando diferentes entidades. Esto hace que sea necesario evaluar el contexto en el cual el comando o expresión aparece.

Toda definición produce una asociación o enlace entre el identificador declarado y la entidad a la que hará referencia. En la mayoría de los lenguajes de programación, las definiciones tienen un ámbito, que es la porción del programa en la cual dicha definición es efectiva.

Un entorno es un conjunto de enlaces. Cada expresión será interpretada en un entorno en particular, y todos los identificadores que aparezcan en ella deberán estar enlazados en dicho entorno.

Pasaje de Parámetros [6]

Un argumento es un valor o cualquier otra entidad que se pasa a un procedimiento o función. Un parámetro es una expresión u otro dato que produce un argumento.

Un parámetro formal es un identificador por el cual el procedimiento tiene acceso al argumento. De esta manera, parámetro y argumento están íntimamente relacionados, y a pesar de sus diferencias específicas, se las suelte tomar como sinónimos. En general, todos los valores pueden ser pasados como argumentos. En algunos, pueden pasarse. En algunos lenguajes, también otro tipo de expresiones como funciones o punteros también pueden ser pasados como argumentos.

Cuando un procedimiento es invocado, cada parámetro formal quedará asociado con su correspondiente argumento. Esto se conoce como mecanismo de pasaje de parámetros y se puede clasificar en:

- Pasaje de parámetros por **copia**: mecanismo por el cual se unifica el parámetro formal a una variable local que contiene una copia del argumento.
- Pasaje de parámetros por **referencia**: mecanismo por el cual se unifica el parámetro formal al argumento en sí.

Por copia

El mecanismo de pasaje de parámetros por copia permite que un valor sea copiado a la entrada y/o a la salida de un procedimiento. El parámetro formal representa una variable local del procedimiento. Un valor es copiado en el parámetro formal cuando se lo invoca y/o copiado al parámetro formal (debe ser una variable) cuando termina. La variable local es creada, utilizada durante la ejecución del procedimiento y luego destruida.

Existen formas de subclasificar los mecanismos de copia de parámetros:

[6] Una ampliación de estos conceptos se puede encontrar en WATT, David. *Programming Languajes Concepts and Paradigms*.. Capítulo 5.

- Por **valor**: Cuando un procedimiento es invocado, se crea y inicializa una variable local dentro del espacio de memoria del procedimiento con el valor del argumento. Dentro de este entorno la variable puede ser consultado y aún más puede ser modificada. Sin embargo cualquier cambio en dicha variable no tendrá efecto fuera del procedimiento.
- Por **resultado**: En este caso el argumento debe ser una variable. Cuando se invoca al procedimiento, se crea una variable local, pero no se inicializa. Cuando el procedimiento culmina, el resultado final de la variable local es asignado a la variable pasada como argumento.
- Por **valor – resultado**: Es una combinación de los dos casos mencionados anteriormente. En este mecanismo el argumento debe ser también una variable. Cuando se invoca el procedimiento, se crea y inicializa una variable local con el actual valor del argumento. Cuando el procedimiento termina, el valor final de la variable local se asigna a la variable pasada como argumento.

Mecanismos de pasaje de parámetros por copia (PF: Parámetro formal)

Mecanismos	Argumento	Efecto al invocar	Efecto al finalizar
valor	Valor	PF ⇓ valor argumento	---
resultado	Variable	---	Variable argumento ⇓ PF
valor – resultado	Variable	PF ⇓ valor argumento	Variable argumento ⇓ PF

Por referencia

El pasaje de parámetros por referencia permite que el parámetro formal se asocie directamente al argumento en sí.

El pasaje de parámetros por referencia tiene una semántica mucho más simple y es adecuada para cualquier tipo de valor: pueden ser valores constantes, variables u otros procedimientos, pero en todo caso se tendrán que tratar como tales dentro del procedimiento. Este mecanismo se apoya en el acceso indirecto a los datos. Algunos lenguajes determinan el pasaje por copia para datos simples, pero permiten al compilador la elección entre copia o referencia para los datos compuestos.

Un problema de los parámetros por referencia con variables es lo que conoce como "aliasing", que ocurre cuando dos o más identificadores simultáneamente se asocian a una misma variable (o un identificador se asocia a un dato compuesto y otro a uno de sus componentes) y al modificarlos se producen consecuencias no deseadas. Casi siempre la suposición de que diferentes identificadores denotan distintas variables es justificada, pero en presencia del aliasing esta suposición no es correcta.

Mecanismos de evaluación

Los parámetros que se utilizan en la invocación a una procedimiento puede variar en el orden o el momento en que son evaluadas. Con este criterio podemos distinguir dos formas de evaluación: evaluación ansiosa o evaluación diferida.

La diferencia es visible cuando se utilizan expresiones o invocaciones anidadas a procedimientos, donde los argumentos no son valores simples sino expresiones más complejas que requieren evaluación.

Evaluación ansiosa

La evaluación ansiosa, que al ser la más utilizada en los lenguajes de programación, se la asume como la forma natural de evaluación y no se suele identificar con este nombre, consiste en que los argumentos son **evaluados antes de invocar** al procedimiento.

Evaluación diferida

La evaluación diferida, o evaluación no estricta, plantea que los argumentos de una función son recién **evaluados en el momento en que son utilizados** por el procedimiento.

Provoca que se demore o difiera la evaluación de una expresión, permitiendo que en algunos casos, de acuerdo a cómo esté implementado el procedimiento, no sea necesario evaluarla nunca. Así, dicha expresión puede contener errores o provocar ciclos infinitos sin que ello signifique el mal funcionamiento del programa, al contrario, es utilizado como una herramienta para resolver ciertos problemas. Por ejemplo en los lenguajes funcionales, permite la utilización de listas infinitas.

Capítulo 4

Paradigma Imperativo

- **Clasificación**
- **Definición**
- **Principales características**
  - Celdas de memoria
  - Asignación
  - Algoritmos
- **Programación estructurada**
  - Estructuras básicas de control
  - Modularización
- **Historia**
- **Lenguajes**

Clasificación

Al paradigma imperativo se lo enmarca dentro de los paradigmas **procedimentales**.

Definición

El paradigma imperativo debe su nombre al papel dominante que desempeñan las sentencias imperativas, es decir aquellas que indican realizar una determinada **operación que modifica los datos guardados en memoria**. Su esencia es resolver un problema complejo mediante la ejecución repetitiva y paso a paso de operaciones y cálculos sencillos con la asignación de los valores calculados a posiciones de memoria.

La programación en este paradigma consiste en **determinar qué datos son requeridos para el cálculo**, asociar a esos datos una dirección de memoria, y efectuar, paso a paso, una secuencia de transformaciones en los datos almacenados, de forma que el estado final represente el resultado correcto.

Los conceptos básicos del paradigma son representados por la **arquitectura Von Neumann**, ya que utiliza este modelo de máquina para conceptualizar las soluciones.

Principales características

Estas características no son en general exclusivas del paradigma imperativo, sino que también son implementadas por lenguajes de paradigmas procedimentales no imperativos, pero la importancia preponderante que tienen en la forma de resolver los problemas de manera imperativa, hacen que se constituyan como lo característico de este paradigma.[7]

Celdas de memoria

El principal componente de la arquitectura es la memoria, compuesta por un gran número de **celdas que contienen los datos**. Las celdas, llamadas variables, tienen nombres que las referencian y se organizan en diferentes estructuras de datos.

Asignación

Estrechamente ligado a la arquitectura de memoria se encuentra la idea de que **cada valor calculado debe ser almacenado**, o sea, asignado, en una variable. Estas asignaciones se realizan repetitivamente sobre la misma celda de memoria, remplazando los valores anteriores. La asignación determina el estado de una variable, que consiste en el valor que contiene en un momento en particular. Por lo tanto **los resultados corresponden al estado final que asumen las variables**.

Algoritmos

Un programa imperativo, normalmente realiza su tarea ejecutando repetidamente una **secuencia de pasos elementales**, ya que en este modelo computacional la única forma de ejecutar algo complejo es repitiendo una secuencia de instrucciones.

La programación requiere la construcción de "algoritmos", a modo de receta, método, técnica, procedimiento o rutina, que se definen como **conjuntos finito de sentencias**, ordenadas de acuerdo a sus correspondientes estructuras de control, que marcan el flujo de ejecución de operaciones para resolver un problema específico.

[7] Ver las características de los paradigmas procedimentales en el capítulo 1 “Paradigmas de Programación”

Programación estructurada

La programación estructurada surge como un conjunto de técnicas para facilitar el desarrollo de sistemas en lenguajes del paradigma imperativo, pero presenta ideas que también fueron tenidas en cuenta en lenguajes de otros paradigmas.

Estructuras básicas de control

La base teórica de la programación estructurada plantea que cualquier programa, por más grande y complejo que fuera, puede representarse mediante tres tipos de estructuras de control:

- Secuencia
- Selección
- Iteración.

Llamamos bloques de **secuencia** a los bloques de código que se ejecutan secuencialmente, es decir, una instrucción a continuación de la otra.

Los bloques de **selección** son aquellos que bifurcan el flujo de la ejecución del programa según se cumpla cierta condición. Pueden ser decisiones simples con dos posibles salidas del estilo *if-else o* decisiones con múltiples casos como *switch o case.*

Los bloques de **iteración** son también llamados bucles, y permiten repetir un grupo de operaciones varias veces, ya sea de acuerdo a que se cumplan condiciones, con instrucciones como *while,* determinando una cantidad exacta de operaciones, como *for*, o con otros controles similares.

De esta manera, se evita el uso de "saltos", del estilo del *goto*, que alteren arbitrariamente el flujo de ejecución, adelantándose o retrasándose en el código del programa.

Modularización

También propone desarrollar el programa en forma TOP-DOWN, de arriba hacia abajo. Es decir, modularizar el programa **creando porciones más pequeñas de programas con tareas específicas**, que se subdividen a su vez en otros "subprogramas", cada vez más pequeños y operativos. Estos subprogramas, denominados rutinas, funciones, procedimientos, módulos y de otras maneras en los diferentes lenguajes, deben tener sólo un punto de entrada y uno de salida, y realizar conceptualmente una sola tarea.

Historia

Poco tiempo después de la aparición de las primeras computadoras, se comenzó a establecer la comunicación hombre máquina mediante el desarrollo de programas. En un comienzo fueron escritos en lenguaje de máquina y los programadores daban las órdenes al computador en una manera sumamente engorrosa, poco intuitivo, de difícil control. Para superar ésta problemática, en la década del 50 se fueron desarrollado lenguajes de programación, con su propia semántica y sintaxis.

Los primeros en surgir fueron los denominados de bajo nivel ("assembler") que requerían la intervención de un ensamblador para ser traducidos a lenguaje de máquina. Posteriormente fueron desarrollándose los lenguajes de algo nivel y sus correspondientes compiladores o intérpretes.

Los lenguajes de programación de alto nivel iniciales fueron **FORTRAN** y **COBOL**. Fortran introdujo las expresiones simbólicas y los subprogramas con parámetros, mientras que Cobol introdujo el concepto de la descripción de datos. Luego, **ALGOL-60** aportó el concepto de estructura de bloque, donde variables y procedimientos podían declararse en cualquier lugar del programa donde se los necesitara.

Una forma de ganar poder de expresión fue elegir un adecuado conjunto de conceptos y combinarlos sistemáticamente. En **ALGOL-68**, el programador puede definir arreglos de enteros, arreglos de arreglos, arreglos de procedimientos, etc., y de la misma forma definir procedimientos que toman o devuelven enteros, arreglos, procedimientos, etc. **PASCAL** fue durante cierto tiempo el más popular de los lenguajes porque era simple, sistemático y eficientemente implementable. Pascal y Algol-68 estuvieron entre los primeros lenguajes con estructuras de control ricas, tipos de datos ricos y definiciones de tipos.

ADA introdujo los paquetes y genéricos, designados a la construcción de grandes programas modulares, así como formas de alto nivel de excepciones y concurrencia.

Pueden distinguirse algunos lineamientos en la historia de los lenguajes de programación que apuntan a mayores niveles de abstracción. Las mnemotécnicas y etiquetas simbólicas de los lenguajes ensambladores se abstraen de los códigos de operación y las direcciones de memoria. Las variables y asignaciones se abstraen de las búsquedas y actualizaciones de los almacenamientos. Las estructuras de control se abstraen de los saltos. Los procedimientos se abstraen de las subrutinas. Las estructuras de bloques y módulos logran formas de encapsulamiento, la cual ayuda a hacer programas modulares. Los tipos genéricos abstraen partes del programa de los tipos de valores sobre los cuales operan, fomentando su reusabilidad.

La aparición del lenguaje **C**, luego de pruebas en lenguajes que no llegaron a distribuirse en forma comercial que se llamaron A y B, marcó un hito

importante en los lenguajes imperativos. Con una estructura sólida, con manejo de estructuras de datos y aritmética de apuntadores, se consolidó en diversos campos de aplicación, inclusive como base para el desarrollo de otros lenguajes.

La programación imperativa fue sin duda la dominante hasta mediados de los '90, pero otros lenguajes fueron ganando popularidad rápidamente. En la actualidad, siguen siendo numerosos los sistemas desarrollados en lenguajes imperativos, aunque va perdiendo terreno frente a desarrollos de otros paradigmas.

Lenguajes

Existen numerosos lenguajes del paradigma imperativo, como PASCAL, C, FORTRAN, ALGOL, COBOL, ADA, CLIPPER, FOX, PL/1, etc. Muchos de ellos incluyen extensiones que permiten soportar otros paradigmas, en mayor o menor grado; no obstante, sigue predominando en ellos el modelo de máquina imperativa.

Capítulo 5

Paradigma de objetos

- **Clasificación**
- **Definición**
- **Principales características**
  - Objetos
  - Encapsulamiento
  - Mensajes
  - Clase
  - Polimorfismo
  - Herencia
- **Historia**
- **Lenguajes**

Clasificación

El paradigma de objetos se ubica entre los paradigmas **procedimentales**.

Definición

El paradigma de objetos, o como se lo conoce generalmente, la Programación Orientada a Objetos, se fundamente en concebir a un sistema como un conjunto de entidades que representan al mundo real, los **"objetos"**, que tienen **distribuida la funcionalidad e información** necesaria y que **cooperan entre sí** para el logro de un objetivo común.

La Programación Orientada a Objetos es una "filosofía" de desarrollo de software que permite crear unidades funcionales extensibles y genéricas, de forma que el usuario las pueda aplicar según sus necesidades y de acuerdo con las especificaciones del sistema a desarrollar[8].

[8] ALONSO AMO, F y SEGOVIA PEREZ, F. *Entornos y Metodologías de Programación*. Capítulo 1

La Orientación a Objetos permite una representación más directa del modelo de mundo real, reduciendo fuertemente la transformación radical normal desde los requerimientos del sistema, definidos en términos del usuario, a las especificaciones del sistema, definidas en términos del computador[9].

En otras palabras, el paradigma de objetos pretende:

- Desarrollar los sistemas con modelos **más cercanos a la realidad** que a las especificaciones computacionales.
- Construir componentes de software que sean **reutilizables**.
- Diseñar una implementación de manera que puedan ser **extendidos** y **modificados** con el mínimo impacto en el resto de su estructura.

Si bien los desarrollos hechos en otros paradigmas también pueden tener estas características, el Paradigma de Objetos no sólo provee herramientas para que sea más sencillo de lograr, sino que las considera como intrínsecas de la configuración de sus lenguajes. A su vez, no se descarta las propiedades procedimentales que tienen que ver, por ejemplo, con la asignación en memoria o la utilización de estructuras de control para determinar el flujo de secuencia, sino que manteniendo sus propiedades procedimentales, utiliza estas herramientas dentro de otras más amplias y potentes que lo caracterizan como paradigma.

Principales características

Estructura de desarrollo modular basada en **objetos**, que son definidos a partir de **clases**, como implementación de tipos abstractos de datos.

Encapsulamiento como forma de abstracción que separa las interfaces de las implementaciones de la funcionalidad del sistema (**métodos**) y oculta la información (**variables**).

Mecanismo de envío de **mensajes**, que posibilita la interacción entre los objetos y permite la **delegación** de responsabilidades de unos objetos a otros.

Polimorfismo, basado en el **enlace dinámico[10]**, de forma que las entidades del programa puedan referenciar en tiempo de ejecución a objetos de diferentes clases.

Herencia, que permite que una clase sea definida como una extensión o modificación de otra.

[9] ALONSO AMO, F y SEGOVIA PEREZ, F. *Entornos y Metodologías de Programación*. Capítulo 1
[10] Ver la definición de enlace dinámico en el capítulo 3 "Abstracción de control"

Objetos

Los objetos son abstracciones que **representan las entidades** del mundo real que forman parte del **dominio del problema**, a los **componentes computacionales**, tanto de software como de hardware, y a toda unidad de información que sea necesaria para desarrollar un programa. **"Todo" es pensado como un objeto**. Al implementar los objetos mediante un lenguaje de programación, los atributos que conforman el **estado interno** se denominan **variables** y a las funcionalidades que conforman su **comportamiento** se las llama **métodos**.

Mensajes

Durante la ejecución del programa, **los objetos interactúan** solicitándose servicios y recibiendo respuestas mediante el envío de mensajes. Un mensaje consiste en la **invocación de un método** de un objeto en particular. Un diseño que presenta una adecuada distribución de responsabilidades entre los objetos lleva a que cada objeto haga sólo lo que tiene que hacer y que sólo él lo haga, de manera tal que la resolución de una tarea compleja, por más que sea solicitada a un objeto en particular, es resuelta mediante una múltiple **delegación** de responsabilidades que este objeto hace a otros objetos.

Encapsulamiento

Un objeto **no conoce el funcionamiento interno** de los demás objetos y no lo necesita para poder interactuar con ellos, sino que le es suficiente con **conocer su interfase**, es decir saber la forma en que debe enviarles sus mensajes y cómo va a recibir la respuesta. Ante la modificación de una funcionalidad en particular del sistema, en la medida que su implementación esté encapsulada en un objeto, el impacto que produce su cambio no afectará a los otros objetos que interactúan con él.

Clase

Una clase **describe** completa y detalladamente la **estructura de información** y el **comportamiento** que tendrá **todo objeto de esa clase**, o sea, define el conjunto de variables y de métodos que determinan cómo van a ser y cómo se van a comportar sus objetos.

Polimorfismo

El polimorfismo permite a dos o más objetos distintos tengan definidos **métodos de igual nombre**, pero cada uno con su correspondiente implementación. Un objeto emisor puede comunicarse con cualquiera de estos objetos mandándole un mensaje donde se menciona el nombre del método, y el

objeto receptor ejecutará la implementación que tenga definida, independientemente de la otras implementaciones que tengan los otros objetos. Desde el punto de vista del objeto emisor, tampoco le interesa conocer si ante la misma invocación hecha a diferentes objetos, la forma en que cada uno de ellos ejecutó el método internamente fue la misma o no.

Herencia

La herencia permite que se puedan **definir nuevas clases basadas en clases existentes**, lo cual facilita reutilizar código previamente desarrollado. Si una clase hereda de otra tiene como propios todas sus variables y métodos. La clase derivada puede agregar nuevas variables y métodos y modificar los métodos heredados, redefiniéndolos. Las clases se organizan jerárquicamente, siendo unas subclases de otras, de una manera que optimiza la organización de la información y la implementación de los métodos.

Historia

El paradigma de objetos surge en base al paradigma imperativo, provocando un giro importante en sus principios y consideraciones básicas, en el que más que dejarlos de lado, los reorganiza y capitaliza dentro de un modelo más amplio, con otros conceptos característicos.

Su progreso no fue sencillo, sino que llevó muchos años. Smalltalk es considerado el primero de los lenguajes orientados a objetos y el emblemático de este paradigma. En Smalltalk, todo es un objeto, incluso los números enteros. Se basó en ideas de Simula (un lenguaje de simulaciones), pero no es sólo un lenguaje, sino un entorno completo, prácticamente un sistema operativo que se ejecuta encima de una "máquina virtual", lo que asegura su máxima portabilidad entre plataformas.

A pesar de ser un lenguaje simple, poderoso y que promueve buenas prácticas de programación, Smalltalk no llegó a ser un lenguaje muy popular. Esto se debe a la poca aceptación de lenguajes interpretados en los años 1980 y 1990. A pesar de esto algunas grandes empresas llegaron a tener relativo éxito con la plataforma como una herramienta de desarrollo rápido que competía contra herramientas como Power Builder y en menor grado contra Visual Basic que eran muy populares en los años 80 y 90. En 1995, se lanzó Java que es un lenguaje fuertemente influenciado por Smalltalk y que popularizó el concepto de lenguajes interpretados con recolectores de basura y fue diseñado con el propósito de crear un lenguaje que pudiera funcionar en redes computacionales heterogéneas y que fuera independiente de la plataforma en la que se vaya a ejecutar. Esto significa que un programa de Java puede ejecutarse en cualquier máquina o plataforma, con solo tener un intérprete en la misma.

La programación de objetos fue ganando campos de aplicación más velozmente desde mediados de los '90, Y en la actualidad, aunque siguen siendo numerosos los sistemas desarrollados en lenguajes de otros paradigmas la tendencia marca un preferncia por los desarrollos que basan en lenguajes orientados a objetos, especialmente los nuevos desarrollos. En muchos casos, antes de seguir manteniendo y actualizando sistemas en otros lenguajes, se opta por migrarlos a soluciones en lenguajes orientados a objetos.

Lenguajes

El lenguaje originario y paradigmático de la programación en objetos es SMALLTALK. Actualmente existen otros lenguajes más conocidos, como JAVA. Hay también algunos lenguajes como el C++ o el EIFFEL, VISUAL BASIC que son extensiones de otros lenguajes que fueron diseñados básicamente como imperativos, pero que tienen extensiones que incorporan, en mayor o menor medida, los principios del paradigma de objetos

Capítulo 6

Paradigma Funcional

- **Clasificación**
- **Definición**
- **Principales características**
  - Transparencia referencial
  - Evaluación diferida
  - Recursividad
  - Listas
  - Funciones de orden superior
- **Campo de aplicación**
- **Historia**
- **Lenguajes**

Clasificación

El paradigma funcional está encuadrado dentro de los paradigmas **declarativos**. [11]

Definición

El paradigma funcional está basado en el **modelo matemático de composición funcional**. En este modelo, el resultado de un cálculo es la entrada del siguiente, y así sucesivamente hasta que una composición produce el resultado deseado. Así, un programa es un conjunto de funciones que cooperan entre ellas para el logro de un objetivo común.

[11] Existen elementos que hacen a la identidad del paradigma que son consecuencia de ser un paradigma declarativo. Ver clasificación de los paradigmas en el Capítulo 1 "Paradigmas de programación". Los lenguajes que implementan las características de este paradigma, en general incluyen algún tipo de estrategias, como por ejemplo secuencias de control, que no son estrictamente declarativas, por lo que suelen decir también que el paradigma funcional es "seudodeclarativo"

Las **funciones**, basadas en el concepto matemático del término, son expresiones que establecen una relación de correspondencia o asociación entre miembros de un conjunto (el dominio) y miembros de otro conjunto (la imagen), en un determinado sentido. De esta manera, para cada elemento del conjunto dominio corresponde un determinado elemento del conjunto imagen, y no existen elementos del conjunto dominio que no tengan su correspondiente imagen. Generalizando, las funciones tienen su dominio conformado por varios conjuntos y existe un elemento en la imagen para cada elemento del producto cartesiano de todos ellos.

La definición de una función debe contemplar, ya sea con expresiones para casos puntualmente o genéricas, todos las entradas posibles y asociar cada una de ellas con otra expresión que corresponde al resultado. Cuando la función es invocada, los argumentos recibidos se unifican con las expresiones codificadas y se retorna el resultado de su evaluación.

Principales características

Transparencia referencial

Por ser un paradigma declarativo, implementa la transparencia referencial[12] en sus expresiones y no tiene asignación destructiva. **El valor de cada expresión depende únicamente de sus componentes.** Las variables son utilizadas para hacer referencia a valores intermedios y parámetros de las funciones, como resultados de cálculos anteriores y entradas a subsiguientes cálculos. Si bien internamente puedan utilizar alguna porción de memoria, no son "celdas" en las que se vayan realizando sucesivas "asignaciones". Tampoco existen algoritmos, sentencias, comandos ni estructuras de control imperativas.

Evaluación diferida

Para la evaluación de los argumentos de las funciones utiliza el sistema de evaluación diferida[13], por lo que la evaluación de **las expresiones invocadas se posterga hasta el momento en que realmente sean utilizadas**.

Recursividad

Otra característica, lejos de ser exclusiva del paradigma pero que tiene gran importancia en la formulación de soluciones, es la recursividad, que se ve expresada en el uso permanente de **tipos de datos y funciones recursivas**. La recursividad, entendida como iteración con asignación no destructiva, está

[12] Ver definición de transparencia referencial en el Capítulo 2 "Abstracción de datos"
[13] Ver definición de evaluación diferida en el Capítulo 3 "Abstracción de control"

relacionada con el principio de **inducción** y surge de la definición axiomática de los números naturales[14]. En general, una función recursiva se define con al menos un término recursivo, en el que se vuelve a invocar la función que se está definiendo, y algún término no recursivo como caso base para detener la recursividad.

Listas

Entre las estructuras de datos que utiliza se destacan las listas, como forma de organizar **conjuntos de valores**. Están definidas en forma recursiva. Como caso particular, y gracias a la utilización de la evaluación diferida, las listas pueden ser infinitas, es decir, contener tantos elementos como se requiera sin establecer un máximo. Su construcción se suele realizar con funciones recursivas en las que no se definen casos bases que corten la recursividad y permitan así una invocación recursiva infinita.

Funciones de orden superior

La programación funcional incorpora el concepto de función como objeto de primera clase, lo que significa que **las funciones son tratadas como datos** y en consecuencia pueden ser pasadas como parámetros, calculadas, devueltas como resultados o mezcladas en cálculos más complejos con otros datos. Las funciones que reciben a otras funciones como parámetros se llaman funciones de orden superior.

Campo de aplicación

Los creadores de los primeros lenguajes funcionales pretendían convertirlos en lenguajes de uso universal para el procesamiento de datos en todo tipo de aplicaciones, pero con el paso del tiempo, su utilización está relacionada principalmente en ámbitos de **investigación científica** y **aplicaciones matemáticas**.

Historia

La arquitectura del computador ejerce una influencia estableciendo restricciones a los métodos de diseño de soluciones, tendiendo a limitar el lenguaje para que éste se pueda implementar eficientemente en las máquinas existentes. Durante años los lenguajes han estado restringidos por las ideas de

[14] Teorema planteado por Giuseppe Peano en su trabajo de sistematización de las matemáticas con una notación funcional

Von Neumann, ya que la mayoría de las computadoras tienen una estructura muy similar a la original de éste.

A principios de la década de los setenta aparecieron los primeros síntomas de lo que se ha denominado **crisis del software**. Los programadores que se enfrentan a la construcción de grandes sistemas de software observan que sus productos no son fiables. La alta tasa de errores conocidos (*bugs)* o por conocer pone en peligro la confianza que los usuarios depositan en sus sistemas. La raíz del problema radica en la dificultad de **demostrar que el sistema cumple los requisitos** especificados. La verificación formal de programas es una técnica costosa que en raras ocasiones se aplica.

Una posible solución fue proponer un **modelo de computación diferente** al modelo imperativo tradicional**.** Se basa en la idea de que los problemas detectados son inherentes al modelo computacional utilizado y su solución no se encontrará a menos que se utilice un modelo diferente. Así nace la **programación funcional** o aplicativa, cuyo objetivo es describir los problemas mediante funciones matemáticas puras sin efectos laterales

Sin embargo, se observa que a pesar de los años transcurridos, los lenguajes de programación funcional han tenido éxito a la hora de reemplazar a los lenguajes convencionales en ciertas áreas de aplicación, pero no han logrado ubicarse masivamente en aplicaciones de uso general.

Lenguajes

El lenguaje más típico del paradigma funcional es el LISP y existen otros lenguajes similares más modernos como GOFER y HASKELL.

Capítulo 7

Paradigma Lógico

- **Clasificación**
- **Definición**
- **Principales características**
  - Lógica Proposicional
  - Declaraciones
  - Inversibilidad
  - Backtracking
- **Campo de aplicación**
- **Historia**
- **Lenguajes**

Clasificación

El paradigma lógico forma parte de los paradigmas **declarativos**[15].

Definición

El paradigma lógico tiene como característica principal la aplicación de las **reglas de la lógica** para inferir conclusiones a partir de datos. Conociendo la información y las condiciones del problema, la ejecución de un programa consiste en la búsqueda de un objetivo dentro de las declaraciones realizadas. Esta forma de tratamiento de la información permite pensar la existencia de "programas inteligentes" que puedan responder, no por tener en la base de datos todos los conocimientos, sino por poder inferirlos a través de la **deducción**.

Un programa lógico contiene una **base de conocimiento** sobre la que se hacen **consultas**. La base de conocimiento está formada por **hechos**, que representan la información del sistema expresada como relaciones entre datos,

[15] Existen elementos que hacen a la identidad del paradigma que son consecuencia de ser un paradigma declarativo. Ver clasificación de los paradigmas en el Capítulo 1 "Paradigmas de programación".

y **reglas lógicas** que permiten deducir consecuencias a partir de combinaciones entre los hechos y, en general, otras reglas. Se construye especificando la información del problema real en una base de conocimiento en un lenguaje formal y el problema se resuelve mediante un mecanismo de inferencia que actúa sobre ella. Así pues, una clave de la programación lógica es poder expresar apropiadamente todos los hechos y reglas necesarios que definen el dominio de un problema.

Principales características

Lógica Proposicional

El paradigma tiene sus fundamentos en las teorías de la lógica proposicional. De ella, se toma un tipo especial de lógica conocido como "lógica de predicados de primer orden" y más en particular aún, las **Cláusulas de Horn**. Dichas cláusulas son una forma de lógica de predicados con una sola conclusión en cada cláusula y un conjunto de premisas de cuyo valor de verdad se deduce el valor de verdad de la conclusión: **una conclusión es cierta si lo son simultáneamente todas sus premisas.**

Declaraciones

Por su esencia declarativa, un programa lógico no tiene un algoritmo que indique los pasos que detallen la manera de llegar a un resultado. Es el sistema el que proporciona la secuencia de control, por lo que el programa se transforma en un conjunto de **declaraciones formales que describen la solución**, que deben ser correctas por definición, por lo que la corrección del programa debería estar probada automáticamente. No existe el concepto de asignación a celdas de memoria, sino que las variables son "unificadas" con valores particulares temporalmente y se van sustituyendo durante la ejecución del programa.

Backtracking

Internamente, existe un **mecanismo interno** llamado backtracking que actúa como **control de secuencia**. Durante la ejecución de un programa va evaluando y combinando las reglas lógicas de la base de conocimiento para lograr los resultados esperados. El backtracking no termina la búsqueda de soluciones apenas logra una respuesta válida, sino que agota todas las combinaciones lógicas posibles para ofrecer como el conjunto completo de respuestas alternativas posibles a toda consulta efectuada. Su estrategia es buscar primero en profundidad.

Inversibilidad

En otros paradigmas, las salidas son funcionalmente dependientes de las entradas, por lo que el programa puede verse abstractamente como la implementación de una transformación de entradas en salidas. La *programación lógica* está basada en la noción de que el programa implementa una relación, en vez de una transformación. Los predicados son relaciones, que al no tener predefinido una "dirección" entre sus componentes, permiten que **sus argumentos actúen indistintamente como argumentos de entrada y salida**. Esta característica se denomina como reversibilidad o inversibilidad. Dado que las relaciones son más generales que las transformaciones, la programación lógica es potencialmente de más alto nivel que la de otros paradigmas.

Campo de aplicación

El paradigma es ampliamente utilizado en las aplicaciones que tienen que ver con la **Inteligencia Artificial**, particularmente en el campo de sistemas expertos y procesamiento del lenguaje humano. Un sistema experto es un programa que imita el comportamiento de un experto humano. Por lo tanto contiene información (es decir una base de conocimientos) y una herramienta para comprender las preguntas y encontrar la respuesta correcta examinando la base de datos (un motor de inferencia).

También es útil en **problemas combinatorios** o que requieren gran cantidad o amplitud de soluciones alternativas, dada la naturaleza combinatoria del mecanismo de backtracking.

Historia

La base conceptual de la lógica proposicional es desarrollada por Alfred Horn, en los años 50, en forma independiente al desarrollo computacional, al publicar "Sobre sentencias las cuales son verdaderas de la unión directa de las álgebras", en la cual presenta un modelo lógico para el tratamiento de oraciones del lenguaje natural, donde se explican las luego denominadas "claúsulas de Horn".

En la primer mitad de la década del 70, en base a las cláusulas de Horn, surgen los primeras versiones de lenguajes lógicos (el PROLOG) como una herramienta para resolver ciertos problemas en el área de la inteligencia artificial, originalmente vinculados al tratamiento computacional del lenguaje natural. Luego se va perfeccionando el lenguaje y se escribe el compilador.

Pocos años después se definen los principios que constituyen al Paradigma Lógico como un paradigma de programación, con la participación determinante de Robert Kowalski.

Desde entonces, el lenguaje PROLOG ha sido implementado con diferentes versiones y desarrollado por distintos fabricantes, manteniendo la misma base del lenguaje, pero con pequeñas variantes y herramientas adicionales para facilitar la programación.

Lenguajes

El **PROLOG** es el lenguaje emblemático del paradigma. Existen diferentes versiones y variantes, pero todas basadas en la misma raíz.

Capítulo 8

Paradigma Heurístico

- **Definición**
- **Principales características**
  - Simpleza y velocidad
  - Inexactitud
  - Incorporación del conocimiento
  - Optimización o satisfacción
- **Campo de aplicación**
- **Lenguajes**

Definición

El Paradigma Heurístico se basa en una forma de modelar el problema, en lo que respecta a la representación de su estructura, estrategias de búsqueda y métodos de resolución, mediante **reglas de "buena lógica"** o reglas de "sentido común**"**, denominadas heurísticas[16], las cuales proporcionan entre varios cursos de acción uno que presenta visos de ser el **más "prometedor"**, pero **no garantiza** necesariamente el **curso de acción más efectivo**.

El entorno de programación heurístico es un entorno basado en el conocimiento humano (la experiencia), adaptativo, incremental y simbólico, y aplicable a dominios específicos en los que una buena heurística guía un proceso algorítmico o proporciona resultados superiores a éste.[17]

Principales características

Las especificaciones más relevantes del tratamiento heurístico deben tener en cuenta las características de la heurística, de la información y de las especificaciones del problema:

[16] Una heurística es la conclusión del razonamiento humano en un dominio específico.
[17] ALONSO AMO, F y SEGOVIA PEREZ, F. *Entornos y Metodologías de Programación*. Capítulo 1

Simpleza y velocidad

Una buena heurística debe ser simple, con **velocidad de búsqueda que no se incremente exponencialmente**, precisa y robusta.

Inexactitud

La información a tratar es fundamentalmente inexacta, simbólica o limitada, como también los resultados obtenidos, en los que no se puede garantizar un 100% de exactitud.

Incorporación del conocimiento

La información utilizada como criterio para decidir entre los distintos cursos de acción está basada en el conocimiento previo sobre el dominio del problema, y tiene un crecimiento "incremental" a medida que se avanza en la ejecución, **incorporando el conocimiento obtenido durante la búsqueda**.

Optimización o satisfacción

Las especificaciones del problema deben ser claras y pueden ser de **optimización de soluciones previas** o de **satisfacción de nuevos problemas**, y por otro lado, pueden producir una o múltiples soluciones.

Campo de aplicación

Dado que el ser humano opera la mayor parte de las veces utilizando heurística, este tipo de programación se aplica con mayor intensidad en el campo de la **Inteligencia Artificial** y en especial, en el de la **Ingeniería del Conocimiento**.

En aquellos problemas cuya solución implica potencialmente una búsqueda exhaustiva de todas las posibles combinaciones de algún conjunto finito, que puede producir un incremento exponencial del espacio de búsqueda, que lo hace muy difícil de tratar e incluso imposible, las técnicas heurísticas guían la búsqueda de soluciones por las direcciones que parecen ser las más adecuadas, en base al conocimiento específico sobre el problema en particular, y así evitan recorrer todas las posibilidades.

Se suele utilizar un modelo heurístico cuando proporciona resultados superiores a los de otros modelos, en general en los siguientes casos:

- Los datos del modelo son limitados o pueden contener errores inherentes.

- Se construye un modelo simplificado e impreciso de un problema real, por lo que la solución "óptima" es puramente académica.
- No se dispone de un método exacto que sea fiable para ser aplicado en el modelo del problema, o si existe, es intratable computacionalmente.
- Se desea mejorar la eficacia de un algoritmo realizado con técnicas no heurísticas.

Lenguajes

El Paradigma Heurística **no ha producido un lenguaje específico** de programación que soporte todas sus características, pero las técnicas heurísticas, se pueden implementar con lenguajes de otros paradigmas de programación.

Capítulo 9

Paradigma Concurrente

- **Definición**
- **Principales características**
  - Comunicación y sincronización de procesos
  - Competencia y cooperación
  - Paralelismo en hardware
  - Problemas frecuentes a evitar
  - Mecanismos concurrentes
- **Campo de aplicación**
- **Lenguajes**

Definición

El concepto fundamental de la programación concurrente que da sentido a la existencia del paradigma concurrente, diferenciándolo de otros paradigmas, es la noción de **proceso**.

Un proceso corresponde a un cálculo secuencial con **su propio seguimiento de control**. Su seguimiento es la secuencia de puntos en el programa que son alcanzados mientras el control fluye a través del código del programa. [18]

La **ejecución simultánea** de más de un proceso permite que cooperen para resolver un mismo problema y, a la vez, requiere necesariamente que compartan los recursos del sistema y compitan por acceder a ellos. Para lograrlo, se implementan distintas estrategias de interacción entre los procesos.

Principales características:

Comunicación y sincronización de procesos

La interacción entre procesos se basa en mecanismos de comunicación y sincronización.

[18] RAVI y SETHI. *Lenguajes de programación - Conceptos y constructores.* Capítulo 9.

La **comunicación** implica el **intercambio de datos** entre procesos, ya sea por medio de un mensaje implícito o a través de valores de variables compartidas. Una variable es compartida entre procesos si es visible al código de esos procesos.

La **sincronización** relaciona el seguimiento de un proceso con el seguimiento de otro. Si p es un punto en el seguimiento de un proceso P, y q es un punto en el seguimiento de un proceso Q, entonces la sincronización puede usarse para restringir el orden en el cual P alcanza p y Q alcanza q. En otras palabras, la sincronización implica el **intercambio de información de control** entre procesos.

Competencia y cooperación

La necesidad de contar con comunicación y sincronización puede verse en términos de competencia y cooperación entre procesos.

La **competencia** ocurre cuando **el proceso requiere el uso exclusivo de un recurso**, como cuando dos procesos compiten por usar a la vez un mismo dispositivo de hardware, o para realizar una transacción sobre una misma estructura de datos. Aquí la sincronización es necesaria para garantizar a un proceso el uso exclusivo de un recurso.

La **cooperación** ocurre cuando **dos procesos trabajan sobre distintos aspectos del mismo problema**, y por lo general incluye a la comunicación y la sincronización.

Paralelismo en hardware

La **concurrencia en un lenguaje de programación y el paralelismo en el hardware son dos conceptos independientes**. Las operaciones de hardware ocurren en paralelo si ocurren al mismo tiempo. Las operaciones en el texto fuente son concurrentes si pueden ejecutarse en paralelo, aunque no necesariamente deben ejecutarse así. Se dice que las operaciones que ocurren una después de otra, ordenadas en el tiempo, son secuenciales o serializables. Podemos tener concurrencia en un lenguaje sin hardware paralelo, así como ejecución en paralelo sin concurrencia en el lenguaje. En síntesis, la concurrencia se refiere al potencial para el paralelismo.

Problemas frecuentes a evitar

Algunas situaciones problemáticas propias de la concurrencia de proceso, que se debe evitar que sucedan en un sistema, entre otros, son el interbloqueo y la inanición.

Un programa concurrente se encuentra en **interbloqueo** (deadlock o "abrazo mortal") si todos los procesos están esperando y no pueden proseguir su

ejecución. Por ejemplo, un programa solicita dos recursos iguales, y uno de ellos sólo es tomado si ya tomó el otro en primer lugar. Dos de estos procesos entrarían en interbloqueo porque cuando cada uno tome un recurso, no podrá tomar el otro, y no podrán avanzar.

Otra situación es el **estancamiento** (o inanición), en el cual el sistema no está en interbloqueo, pero ningún proceso avanza. Supongamos que el proceso anterior cambia su lógica para pedir dos recursos, y libera el primero si no puede tomar el segundo. Por lo tanto, dos de estos procesos ejecutando concurrentemente entrarían en un ciclo infinito: tomar el primero, liberar el primero, tomar el primero, etc.

Mecanismos concurrentes

Para concretar la sincronización y comunicación entre procesos y garantizar que se complete la ejecución de todos ellos, existen numerosas estrategias y mecanismos.

Algunos de ellos son:

- **Rendezvouz**
- **Monitores**
- **Semáforos**

Campo de aplicación

La programación concurrente tiene aplicación en los más variados campos y se ha potenciado su uso con el crecimiento de las redes de todo tipo. Tradicionalmente, uno de sus utilidades básicas es en el diseño de sistemas operativos.

Lenguajes

No hay en la actualidad lenguajes de programación exclusivos del paradigma concurrente, sino que existen lenguajes propios de otros paradigmas que incluyen en su definición conceptos y servicios aptos para que soporten la concurrencia.

Bibliografía

- **ALONSO AMO, F y SEGOVIA PEREZ, F**. *Entornos y Metodologías de Programación*. Paraninfo.
- **WATT, David**. *Programming Languajes Concepts and Paradigms*. Prentice Hall.
- **GHEZZI, Carlo y JAZAYERI, Mehdi**. *Conceptos de Lenguajes de Programación*. Díaz de los Santos.
- **RAVI y SETHI.** *Lenguajes de programación - Conceptos y constructores.* Addison Wesley.

Índice